Simulador de CPU - Arquitetura e Organização de Computadores

Este projeto visa implementar um simulador de uma CPU baseada na arquitetura de von Neumann, a fim de melhor compreender o funcionamento interno de um processador. O simulador executa programas escritos em um formato binário simples, o ciclo da Máquina de Von Neumann que pode ser descrito nas operações: busca, decodifica e executa. Ao final de cada ciclo, o estado da CPU e da memória são exibido no terminal, aguardando que o usuário pressione uma tecla para prosseguir.

Como Compilar e Executar

O projeto é composto por múltiplos arquivos `.c`, incluindo o programa principal e a biblioteca auxiliar `assembler.c`. Para compilar corretamente, é necessário incluir todos eles na linha de compilação.

Compilação

Se estiver usando GCC no terminal:

```
gcc neumannMachine.c assembler.c -o cpu_simulator
```

Isso criará o executável chamado `cpu_simulator`.

Como Executar

Você pode informar o arquivo de entrada (programa `.txt` com o código a ser carregado) de duas maneiras:

1. Passando o nome do arquivo via linha de comando:

```
./cpu_simulator nome_do_programa.txt
```

Substitua `nome_do_programa.txt` pelo nome real do seu arquivo de programa.

---

2. Digitando o nome do arquivo durante a execução:

Se você rodar apenas:

```
./cpu_simulator
```

O programa vai solicitar o nome do arquivo:

```
Please type the name of the .txt file:
```

Basta digitar o nome (exemplo: `programa1.txt`) e pressionar Enter.

Exemplo Completo de Uso

```
gcc neumannMachine.c assembler.c -o cpu_simulator
./cpu_simulator programa1.txt
```

Ou:

```
gcc neumannMachine.c assembler.c -o cpu_simulator
./cpu_simulator
```

(e quando o programa perguntar, digite o nome do arquivo de texto com o código da CPU)

Arquitetura da CPU Simulada

Registradores Arquiteturais

- MBR (Memory Buffer Register)
  Armazena dados sendo lidos ou escritos na memória.
  Tipo: `unsigned int` (32 bits)
- MAR (Memory Address Register)
  Guarda o endereço de memória a ser acessado.
  Tipo: `unsigned short int` (16 bits)
- IR (Instruction Register)
  Armazena o opcode da instrução atual.
  Tipo: `unsigned char` (8 bits)
- RO0 e RO1 (Register Operand 0 e 1)
  Índices dos registradores utilizados pela instrução.
  Tipo: `unsigned char` (8 bits)
- IMM (Immediate)
  Valor imediato para instruções do tipo imediato.
  Tipo: `unsigned short int` (16 bits)
- PC (Program Counter)
  Endereço da próxima instrução a ser buscada na memória.
  Tipo: `unsigned short int` (16 bits)
- E, L, G (Flags)
  - E (Equal) → 1 se os operandos forem iguais.
  - L (Lower) → 1 se o primeiro operando for menor.
  - G (Greater) → 1 se o primeiro operando for maior.
    Tipo: `unsigned char` (8 bits) cada
- Reg[4] (Registradores de Propósito Geral)
  Quatro registradores de uso geral: R0, R1, R2 e R3.
  Tipo: `unsigned short int reg[4]` (16 bits cada)

Memória

- Vetor de 154 posições, com 8 bits cada (1 byte).
  Tipo: `unsigned char memory[154]`

Funcionamento do Simulador

O simulador executa as seguintes etapas para cada ciclo de máquina:

1. Busca
   Busca a instrução na memória no endereço indicado por PC, carrega a palavra de instrução no MBR e extrai o opcode da instrução para o IR.

2. Decodificação
   Decodifica o opcode (IR) e identifica os operandos (RO0, RO1), imediatos (IMM) e endereços (MAR) serão usados.

3. Execute (Execução)
   Executa a operação correspondente, atualizando o conteúdo dos registradores de proposito geral, da memória ou do PC.

Após cada execução, o estado atual da CPU e da memória é exibido no terminal.

Funções Principais

- `main()`
  Gerencia o ciclo de execução da CPU e permite reinicializar o programa após a execução.
- `fetch()`
  Realiza a busca da próxima instrução na memória e carrega no MBR e IR.
- `decode()`
  Decodifica o conteúdo do MBR e extrai os operandos, imediatos ou endereços.
- `execute()`
  Executa a operação de acordo com o opcode (IR).
- `getSourceFile()`
  Lê o código fonte para exibição da linha correspondente no ciclo.
- `getInstruction()` Identifica qual é a linha do código fonte correspondente ao estado atual da cpu.
- `displayCPUStatus()`
  Exibe os valores atuais dos registradores e da memória.

Biblioteca `assembler.h`

Responsável por realizar a montagem (assembly) do código-fonte escrito em linguagem de montagem (assembly-like) para a linguagem de máquina simulada. Esta biblioteca é utilizada pelo programa principal para carregar o conteúdo binário (as instruções e dados) na memória da CPU simulada.

Funções principais da `assembler.h`:

* `loadProgram(char *filename, unsigned char *memory)` Lê um arquivo de texto contendo o programa em linguagem assembly e traduz cada linha para sua representação binária, armazenando diretamente na memória simulada.
* `getOpcode(char *mnemonic)` Faz a tradução de um mnemônico textual (exemplo: `"ld"`, `"add"`, `"jmp"`, etc.) para o valor numérico correspondente ao opcode da arquitetura simulada.

Tratamento de erros

A biblioteca possui verificações para:

* Sintaxe incorreta;
* Tipos inválidos (exemplo: linhas que não são nem dado "d" nem instrução "i");
* Instruções desconhecidas (mnemônicos não reconhecidos);

Em caso de erro, o programa exibe mensagens informativas indicando a linha onde o problema ocorreu e sugere ao usuário verificar a documentação da linguagem para obter mais informações.

Conjunto de Instruções Suportadas

Mnemonic	Opcode (Binary)	Description
`hlt`	`00000`	HALT: Stops the processor. No register is modified. Used at end of program.
`nop`	`00001`	NO OPERATION: Increments PC. No other changes.
`ldr rX, rY`	`00010`	LOAD VIA REGISTER: `rX = rY`
`str rX, rY`	`00011`	STORE VIA REGISTER: `rY = rX`
`add rX, rY`	`00100`	ADD REGISTER: `rX = rX + rY`
`sub rX, rY`	`00101`	SUBTRACT REGISTER: `rX = rX - rY`
`mul rX, rY`	`00110`	MULTIPLY REGISTER: `rX = rX * rY`
`div rX, rY`	`00111`	DIVIDE REGISTER: `rX = rX / rY`
`cmp rX, rY`	`01000`	COMPARE REGISTER: Sets flags E, L, G based on comparison between `rX` and `rY`.
`movr rX, rY`	`01001`	MOVE REGISTER: `rX = rY`
`and rX, rY`	`01010`	LOGICAL AND: `rX = rX & rY`
`or rX, rY`	`01011`	LOGICAL OR: `rX = rX
`xor rX, rY`	`01100`	LOGICAL XOR: `rX = rX ^ rY`

`not rX`	`01101`	LOGICAL NOT: `rX = !rX`
`je Z`	`01110`	JUMP IF EQUAL: If `E == 1`, sets `PC = Z`.
`jne Z`	`01111`	JUMP IF NOT EQUAL: If `E == 0`, sets `PC = Z`.
`jl Z`	`10000`	JUMP IF LOWER: If `L == 1`, sets `PC = Z`.
`jle Z`	`10001`	JUMP IF LOWER OR EQUAL: If `E == 1` or `L == 1`, sets `PC = Z`.
`jg Z`	`10010`	JUMP IF GREATER: If `G == 1`, sets `PC = Z`.
`jge Z`	`10011`	JUMP IF GREATER OR EQUAL: If `E == 1` or `G == 1`, sets `PC = Z`.
`jmp Z`	`10100`	UNCONDITIONAL JUMP: Sets `PC = Z`.
`ld rX, Z`	`10101`	LOAD FROM MEMORY: Loads 16-bit word from memory address `Z` into `rX`.
`st rX, Z`	`10110`	STORE TO MEMORY: Stores 16-bit value from `rX` to memory address `Z`.
`movi rX, IMM`	`10111`	MOVE IMMEDIATE: `rX = IMM`.
`addi rX, IMM`	`11000`	ADD IMMEDIATE: `rX = rX + IMM`.
`subi rX, IMM`	`11001`	SUBTRACT IMMEDIATE: `rX = rX - IMM`.
`muli rX, IMM`	`11010`	MULTIPLY IMMEDIATE: `rX = rX * IMM`.
`divi rX, IMM`	`11011`	DIVIDE IMMEDIATE: `rX = rX / IMM`.
`lsh rX, IMM`	`11100`	LEFT SHIFT: Shifts `rX` left by `IMM` bits.
`rsh rX, IMM`	`11101`	RIGHT SHIFT: Shifts `rX` right by `IMM` bits.

Controle

- `hlt` → Finaliza a execução.
- `nop` → Sem operação.
- `je`, `jne`, `jl`, `jle`, `jg`, `jge`, `jmp` → Instruções de salto.

Aritméticas

- `add`, `sub`, `mul`, `div` → Operações entre registradores.
- `addi`, `subi`, `muli`, `divi` → Operações com valores imediatos.

Lógicas

- `and`, `or`, `xor`, `not`

Transferência

- `ldr`, `str` → Acesso via registrador.
- `ld`, `st` → Acesso via endereço de memória.
- `movr`, `movi` → Movimentação entre registradores ou com imediato.

Comparação

- `cmp` → Compara dois registradores e atualiza as flags E, L e G.

Deslocamento de bits

- `lsh` → Deslocamento à esquerda.
- `rsh` → Deslocamento à direita.

Formato do Arquivo de Programa (.txt)

Os programas a serem carregados na memória da CPU devem ser escritos em um arquivo de texto (`.txt`), obedecendo a seguinte estrutura por linha:

```
[MEMORY_ADDRESS];[TYPE];[CONTENT]
```

Onde:

- `MEMORY_ADDRESS`:
  Endereço de memória (em hexadecimal) onde o conteúdo será armazenado.
- `TYPE`:
  Indica o tipo de linha:
  - `d` → Linha de dado
  - `i` → Linha de instrução
- `CONTENT`:
  - Se for um dado (`d`): Um valor hexadecimal a ser armazenado na memória.
  - Se for uma instrução (`i`): O opcode (em formato de mnemônico, ex.: `ld`, `add`, `jmp`) seguido dos operandos necessários.

Exemplos:

Exemplo de linha com dado:

```
8A;d;0A
```

> Armazena o valor `0A` (hexadecimal) na posição de memória `8A`.

Exemplo de linha com instrução:

```
00;i;ld r0, 8A
```

> Instrução para carregar (`ld`) o conteúdo da posição de memória `8A` no registrador `r0`, sendo armazenada a partir da posição `00` na memória.

- Cada linha do programa representa uma posição específica da memória.
- Instruções são convertidas para o formato binário correto pelo assembler do programa antes de serem carregadas.
- Linhas mal formatadas ou com tipos desconhecidos (qualquer coisa diferente de `i` ou `d`) causarão erro de sintaxe na leitura.

Creditos

Este projeto foi desenvolvido para a disciplina de Arquitetura e Organização de Computadores I do curso de Bacharelado em Ciências da Computação do Instituto Federal de Educação, Ciência e Tecnologia de Goiás - Campus Anápolis. Projeto idealizado Prof. Dr. Eng. Hugo Vinícius Leão e Silva e desenvolvido pelas alunas Karolayne Amábile Brito Borges e Ana Laura Machado Pereira.